Open in app

Parveen Khurana

124 Followers About Following

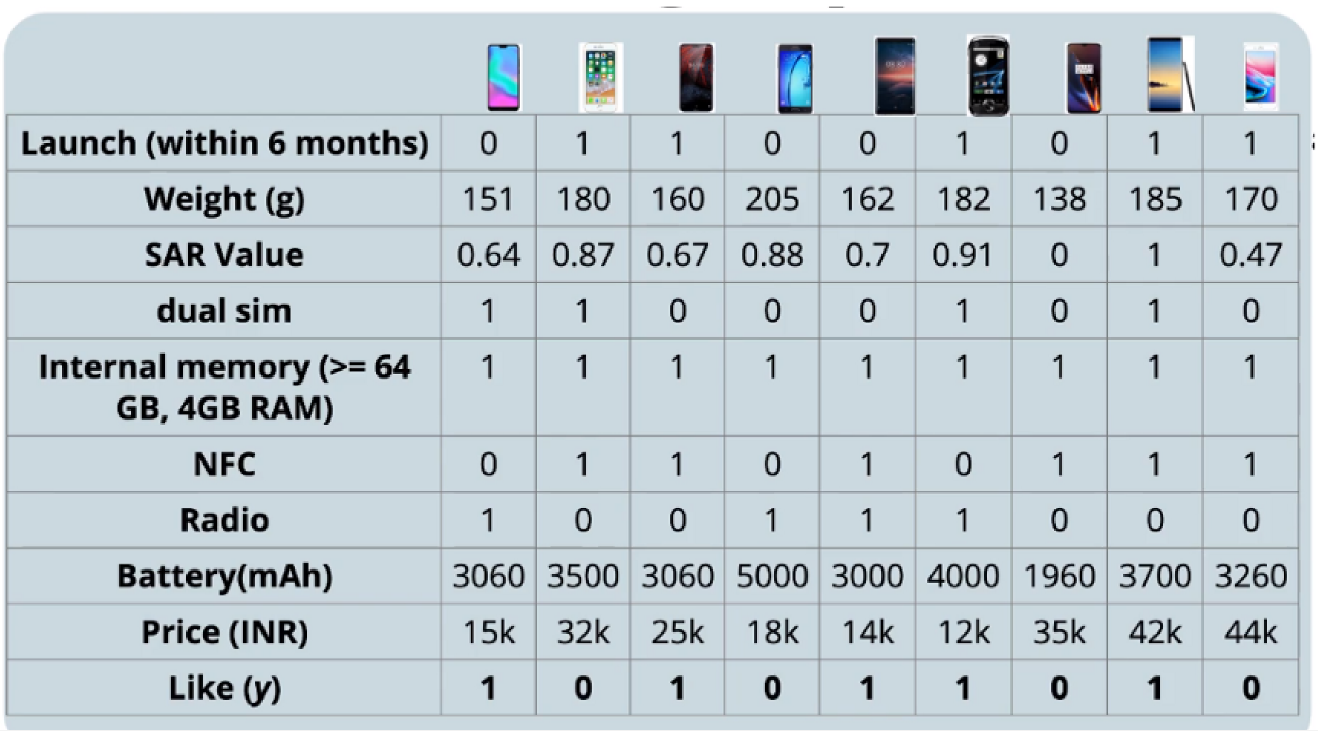
Batch Normalization

Parveen Khurana Feb 23, 2020 · 10 min read

This article covers the content discussed in Batch Normalization and Dropout module of the Deep Learning course and all the images are taken from the same module.

Normalizing Inputs:

There are two terms: one is normalizing the data and the other is standardizing of the data. Normalizing typically means when we do the '**max-min shifting**', in the real-world dataset, we could have different ranges for different **features/attributes**, for example in the below case we have the feature "dual sim" taking on a binary value where the feature "Price" is in 5 digits and so on with other features.



Launch (within 6 months)	0	1	1	0	0	1	0	1	1
Weight (g)	151	180	160	205	162	182	138	185	170
SAR Value	0.64	0.87	0.67	0.88	0.7	0.91	0	1	0.47
dual sim	1	1	0	0	0	1	0	1	0
Internal memory (>= 64 GB, 4GB RAM)	1	1	1	1	1	1	1	1	1
NFC	0	1	1	0	1	0	1	1	1
Radio	1	0	0	1	1	1	0	0	0
Battery(mAh)	3060	3500	3060	5000	3000	4000	1960	3700	3260
Price (INR)	15k	32k	25k	18k	14k	12k	35k	42k	44k
Like (*y*)	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**0**

either normalize or standardize the data.

One way of doing that is we compute the minimum value of every feature and then we shift all the other values of that features by using the minimum value and the maximum value. We subtract the minimum value from each of the values and then divide this by the range(maximum-minimum) of the given data points.

The other way do to this is to make the data to have **zero mean** and **unit variance**. The terms normalization and standardization are used interchangeably as we are converting data into a standard normal distribution. Here, we compute the mean value of each of the features and the variance value for each of the features, we then normalize the values of the feature as per the below formula:

$$x_{ij}^{norm} = \frac{x_{ij} - \mu_j}{\sigma_j}$$

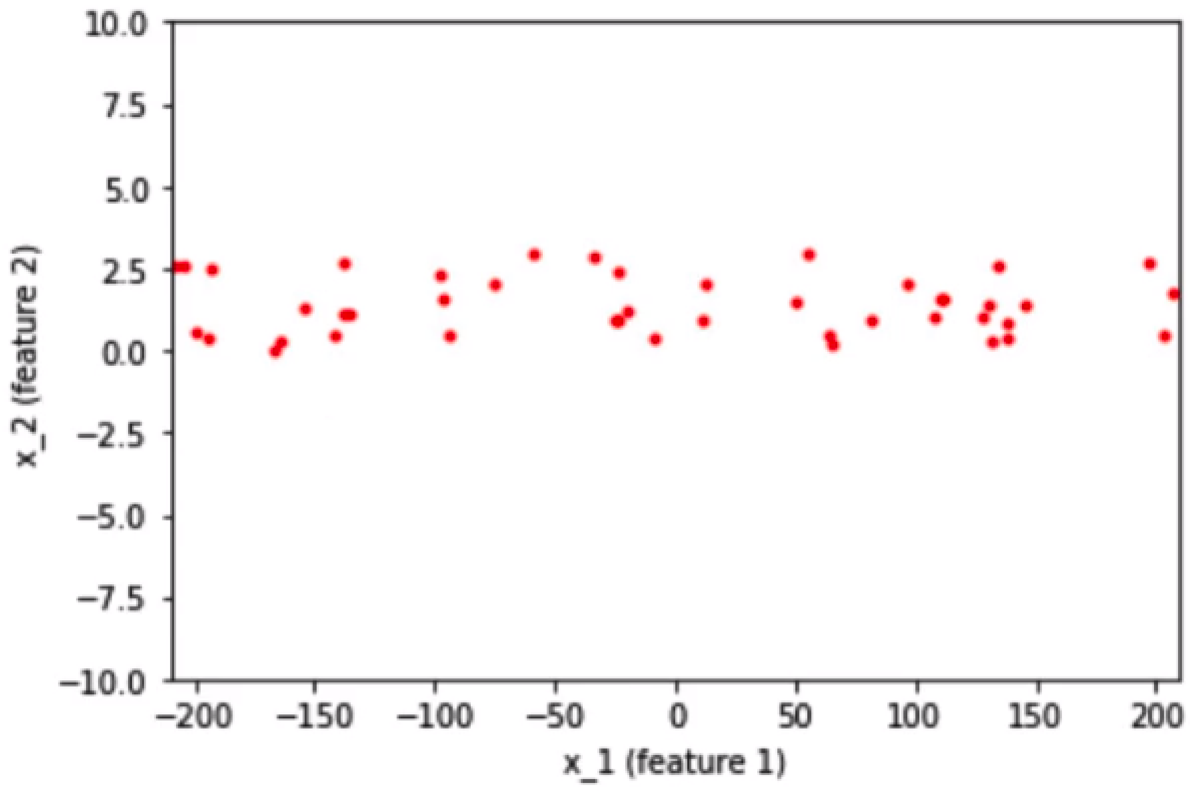
$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_{ij}$$

$$\sigma_j = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(x_{ij} - \mu_j)^2}$$

After applying the changes as per the above formula, all of the values would be close to 0 and would lie in the range of -1 to 1 roughly. This is known as normalizing the data or standardizing the data.

Let’s see why standardizing the data helps us:

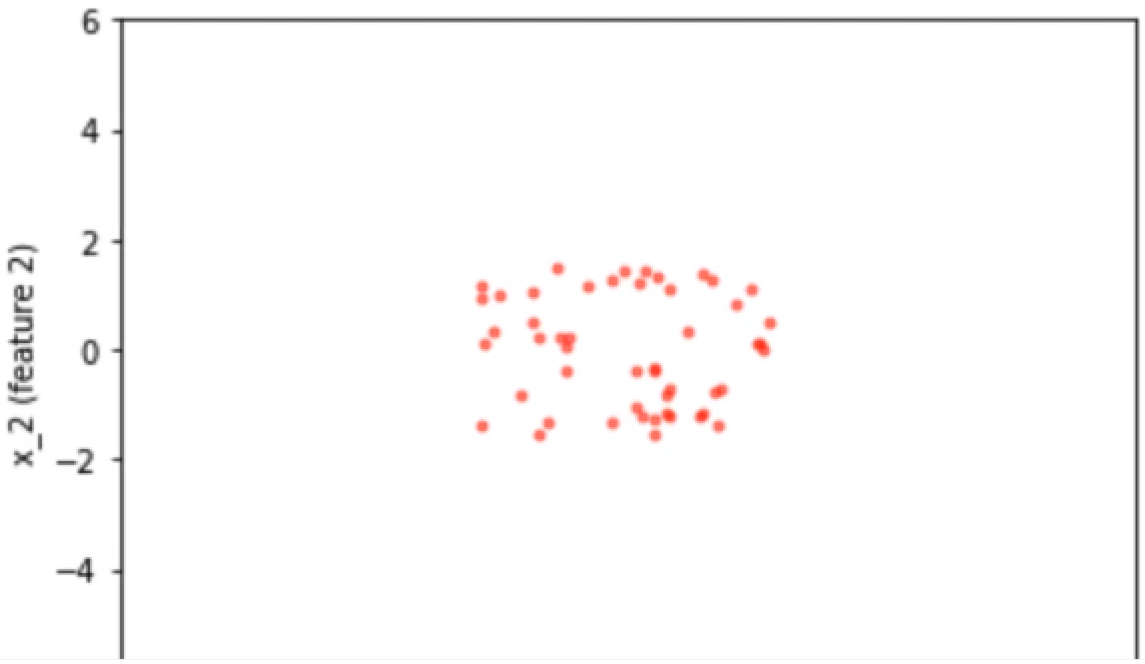
Say we have a situation where we have the below 2D data, we have two features ‘**x1**’ and ‘**x2**’ and one of the features has a wider spread, it ranges from -200 to 200 whereas

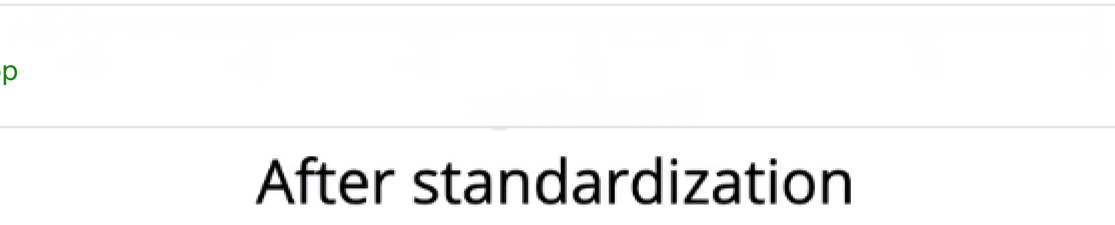


Before standardization

So, the above is the case before data is standardized; if the features are in different ranges then while training the weights would be scaled appropriately.

Let’s see what the data looks like after the standardization:





Now the data lies in the range of -2 to 2 for both the features that we have and it is normally distributed(mean is 0 and variance is 1).

If we don't standardize the data, then the weights would be large for some of the features as per the data and if weights are large then the update in the weights would be large and we might overshoot the minima(Gradient Descent) and if we overshot the minima, then we need to update the weights in the other direction and again this time the update would be large as the weights are large, we might again overshoot the minima and we keep on oscillating like this.

So, in general, if the weights are large, then the updates are large and there would be a lot of oscillations. If the features are in different ranges, then some weights have to be large, some weights have to be small to account for the range difference and for larger weights we would have this problem of over-shooting the minima and the oscillations.

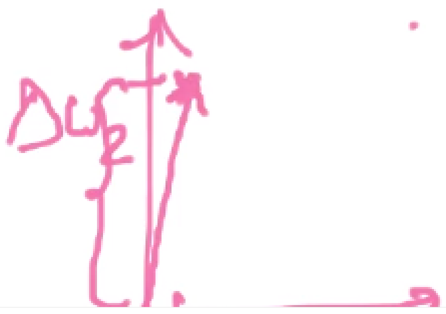
And the way to avoid that is to normalize the weights so that all the weights are in the same range.

The other reason to normalize the data is that if we don't normalize the data then there is a chance that the updates would be biased very much towards the larger weight:

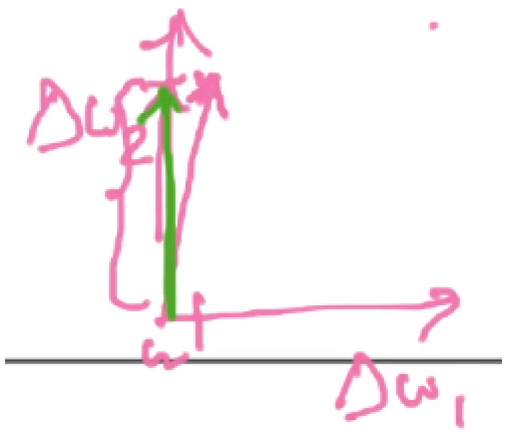
Let's say '**w2**' is larger as compared to the weight '**w1**', if we plot out the '**δw1'** (delta w1) against '**δw2**', then as '**w2**' >> '**w1**' which implies that:

δw2 >> **δw1**

i.e we have a smaller update in the direction of '**w1**' and a larger update in the direction of '**w2**' and the resulting gradient vector is going to look like the below:



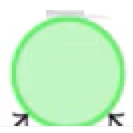
We can see that the resulting vector is biased towards '**w2**', it is **equivalent** to moving in the direction of '**w2**' only(if we moving only in the direction of '**w2**', our vector would be like green vector in the below image) and the pink vector is very close to the green one meaning our updates are more in the direction of '**w2**' or more biased towards the direction corresponding to larger weights and this larger and smaller weights are there because the original data is in different ranges and if not normalized, then weights scale-up appropriately.

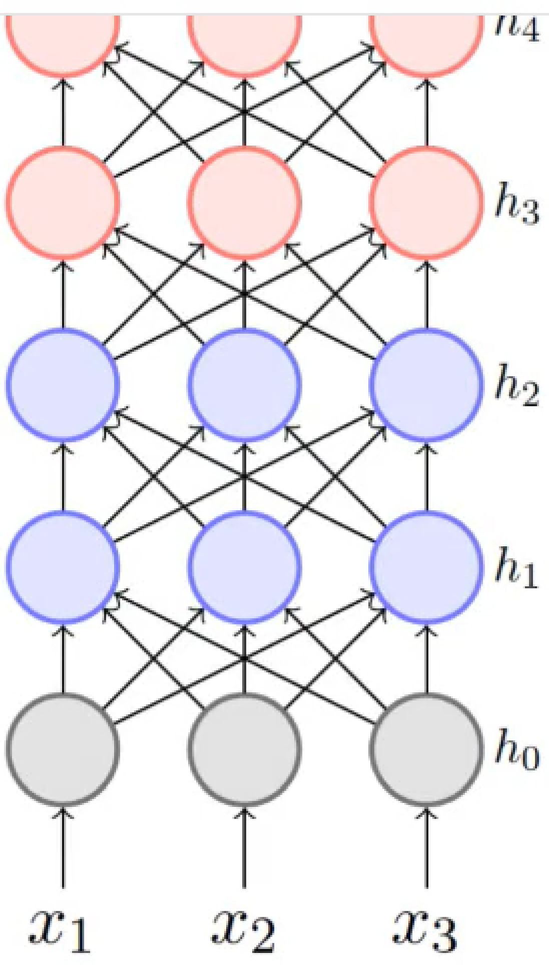


This is why it is not good to have differently scaled weights especially because the model is a linear combination and then off course we are taking a sigmoid on top of that, but in this summation of '$\mathbf{w_i x_i}$', if some of the inputs are large, we need to balance them by having small weights and if some of the inputs are small then we need to have large weights. So, to avoid these problems, we standardize our data.

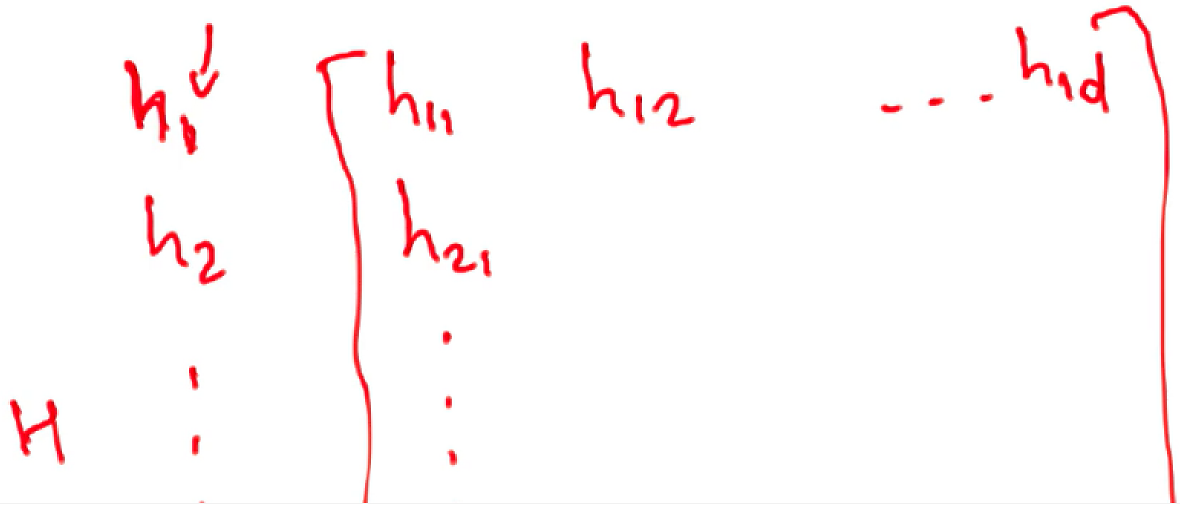
Batch Normalization:

We have so far discussed how to standardize the input data, the next point in the story is, we have some weights(which would be initialized randomly, to begin with during training time) connecting the inputs to the neurons in the intermediate layer and we compute some values(after applying non-linearities at appropriate points) at each layer as we pass the input through the network

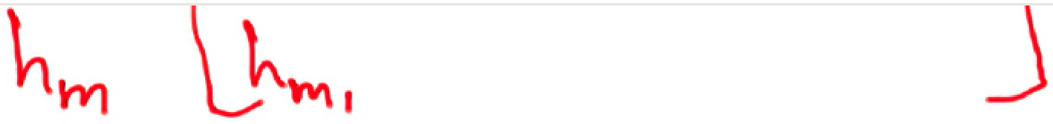




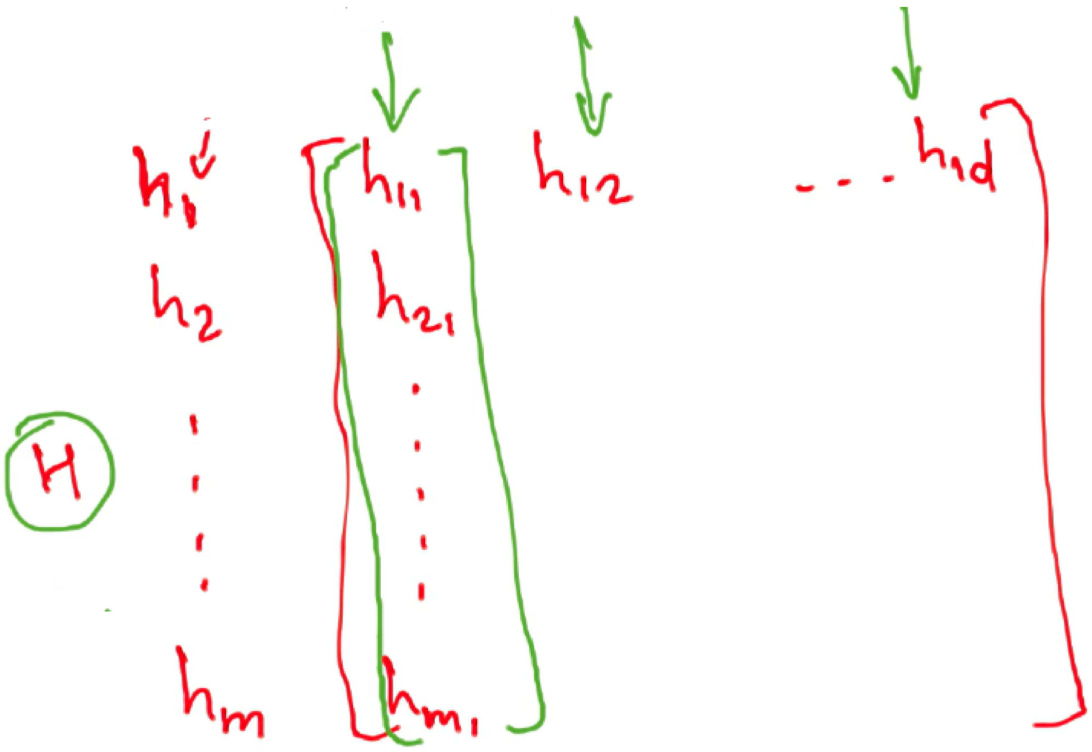
If we look at any particular intermediate layer say ‘**h2**’(let’s call it as ‘**h**’ for now). Let’s say we pass the first data instance ‘**x1**’(which would have 3 values for 3 features that we have in the network) and say we get the output for this data instance as ‘**h1**’, similarly for the second data instance ‘**x2**’, we are calling the output as ‘**h2**’ and so on for the **m’th** data point. Say we have ‘**d**’ dimensions for the output of any intermediate layer, then we can represent the ‘**h**’ matrix like the following:



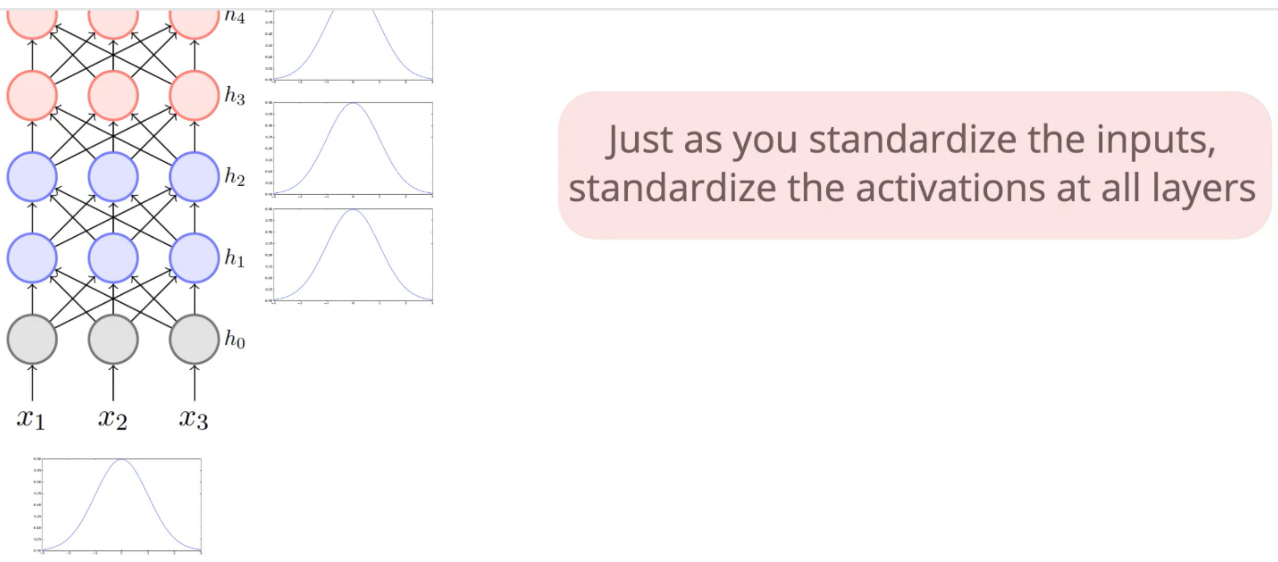
Open in app



Now the second intermediate layer(‘**h2**’, which we are calling as ‘**h**’ in this case) is the input for the next intermediate layer and now the same argument holds that if we want to learn the weights of the next layer effectively, we must standardize the data at this ‘**h2**’ layer.



So, we standardize the data per column/feature/dimension wise at this intermediate layer as well as at all of the other intermediate layers and we standardize using the same formula as for the inputs as discussed above in this article. And this is exactly what we do in batch normalization, just as we standardize the inputs, we standardize the activation values at all of the intermediate layers



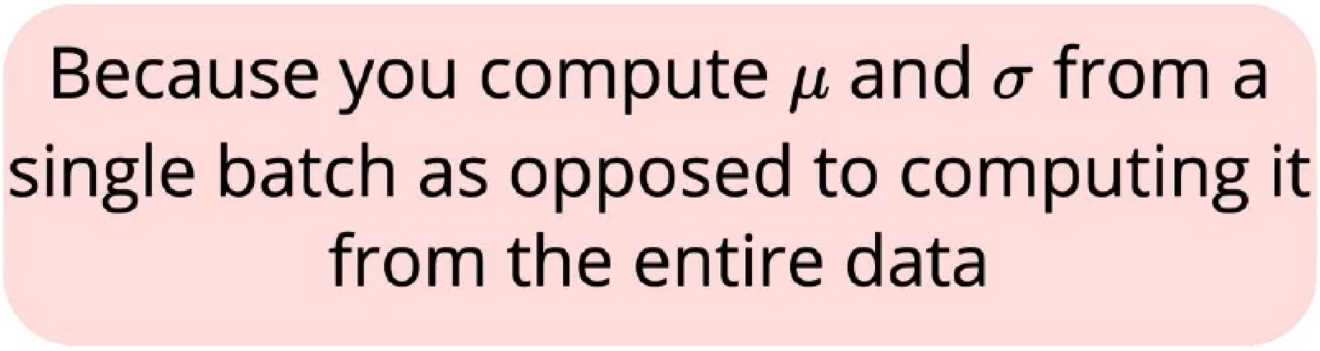
h_4
h_3
h_2
h_1
h_0
x_1
x_2
x_3
Just as you standardize the inputs, standardize the activations at all layers

$$h_{ij} = \frac{\quad}{\sigma_j}$$

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} h_{ij}$$

$$\sigma_j = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(h_{ij} - \mu_j)^2}$$

It is called as Batch Normalization because instead of just having '**m**' data points(considering a total of '**m**' data points in the dataset) in the matrix '**h**', we actually pass the data in batches and that batch might contain just '**k**' data samples where '**k**' < '**m**', and we compute the mean, the standard deviation using the batch size and not the entire dataset and that's why it is called batch normalization as we are computing the metrics based on batch size.


Because you compute μ and σ from a single batch as opposed to computing it from the entire data

Now the question is by normalizing the activations, are we enforcing some constraints on the network and if yes, could that be harmful to the network? The authors of the original paper suggested one more step for the final values of the activations as:

$$h_{ij}^{norm} = \frac{h_{ij} - \mu_j}{\sigma_j}$$

$$h_{ij}^{final} = \gamma_j \cdot h_{ij}^{norm} + \beta_j$$

γ_j and β_j are learned along with other parameters of the network

Gamma and **Beta** are to learned along with the other parameters of the network in the same way as the other parameters meaning there would be update rule for **gamma** and **beta** and this update rule would depend upon the derivative of the loss function with respect to **gamma** and **beta** and we would have one **gamma** and **beta** for every column that we have.

The advantage of this is that let’s say we have two features and in the original data, the variance/spread for one of the features is high compared to the other and which say is good for the final output in some hypothetical situation and by standardizing the data, upon **gamma** and **beta** as the final value of activation depends on **gamma** and **beta**, so now the network has the choice to adjust the values of **gamma** and **beta** in such a way that the data at any of the intermediate layer could have any mean and variance as per the requirement/situation instead of just having zero mean and unit variance.

Suppose for a particular situation, **gamma** becomes equal to **standard deviation** and **beta** becomes equal to the **mean** value, then we have the final values as:

$$h_{ij}^{norm} = \frac{\boxed{h_{ij}} - \mu_j}{\sigma_j} \longrightarrow h_{ij} = \sigma h_{ij}^{norm} + \mu_j$$

$$h_{ij}^{final} = \overset{\nearrow \sigma_j}{\gamma_j} \cdot h_{ij}^{norm} + \overset{\nearrow \mu_j}{\beta_j} \qquad h_{ij}^{f} = \sigma_j \cdot h_{ij}^{norm} + \mu_j$$

As is clear from the above image, if **gamma** is equal to **standard deviation** and **beta** is equal to the **mean** value, then **in that case, the final value of the activation even after the standardization remains equal to the initial value(before standardization) of the activation**. This means that if the **mean** and **variance** of the data before the normalization is good for the overall loss, and we still standardize the data as a step in the process, then, in that case, the network would learn the parameters **gamma** and **beta** in such a way that the final values becomes equal to the original value before the standardization.

In essence, we could say that we standardized the inputs because we knew by logic that standardizing the inputs helps so standardizing the activations would also help but additionally now the model has the flexibility so that we have the parameters **gamma** and **beta** for each of the columns that we have and we learn these parameters such

gamma and **beta** in that way such that final values of the activations are the same as the original value of the activation(before these values were standardized).

So, now the network has the flexibility in both the direction, if normalizing helps then let it be, learn **gamma** as 1 and **beta** as 0 in which case the final activation values would be equal to normalized activations values and if the network needs to stick to the original inputs, then it can learn **gamma** as the same as **standard deviation** value and the **beta** as the **mean** value. All of this depends on the loss value and the update rules for **gamma** and **beta** and hopefully, they will converge to a value which is good for the overall loss.

Just like standardizing the inputs helps in better learning as discussed above in this article, the same reason holds for standardizing the activations and the other reason/explanation for **batch normalization is that it acts as a regularizer**. The **mean** value and **standard deviation** values are computed from a mini-batch and not from the entire data because of which these values of **mean** and **standard deviations** are going to be noisy because it did not come from the entire data but from a small batch and whenever we introduce some noise in the data, it leads to better regularization. For example, say our input from the MNIST dataset is the digit 3

and now if we introduce some noise in the data and pass the below image to the network, then it would result in good training of the network as it has to predict the same correct output but with less information.

So, any sort of noise like in the above image always improves the training because now with the fewer information the network has to predict the right class label, it cannot overfit the data, in this case, it has to rely on other features in the input to ensure that the input image is 3 in the above case.

So, any kind of noise always prevents overfitting and makes the model more robust and by computing the **mean** and the **standard deviation** value based on the batch, we are adding some noise which makes the overall process more robust. So, this is the other reason why batch normalization works well in practice.

Deep Learning Normalization Standardization Artificial Intelligence Neural Networks